COMP 161 — Lecture Notes 14
Search and Sort

In these notes we apply the basic recursive and iterative design template to the problems of searching and sorting.

Structural Recursion and std::vectors

Structural recursions is about making recursive procedures that recurse on the recursive structure of the data. The most basic form of this structure come from having an empty collection BASE CASE and a non-base case where the data is deconstructed into the first element and all of the rest. The "rest" is a smaller version of the original structure. More generally, we can identify any non-recursive smallest size[1] as a base case and deconstruct the non-base case in any way so long as repetition of the decomposition eventually results in a base case[2]. Such structures exists abstractly for just about any collection type. However, not all collections support recursive decomposition or do so in an inefficient manner. The C++ *std::vector* does not support recursive decomposition directly.

[1] one element, two elements, etc

[2] all but the last + last, left half + right half, etc.

Thankfully, any structure with indexed elements can be managed recursively thought the recursive handling of the set of index values. A vector of size s used the integer interval $[0, s)$ for its indices. When the vector is empty, then the interval $[0, 0)$ is also empty[3]. The first of this interval is 0 and the rest is $[1, s)$. Similarly, the last is $s - 1$ and all but the last is $[0, s - 1)$. We can generalize this for any range of contiguous positive integers $[a, b)$. The interval is empty if $a \geq b$. When $a < b$, the first is a and the rest is $[a + 1, b)$.

[3] $[a, b)$ is empty if $a \geq b$

To recursively process a vector we must write a procedure that takes the index interval bounds $first$ and $last$. We don't always need both bounds. Pure functions on vectors can often exclude the last when recursing first to last or last when recursing last to first. Having both, however, provides maximum flexibility. If you need to mutate the vector and doing so changes the size, then you're probably going to need to shift first and last to account for the change in the vector's structure. You also can design with both bounds so that you can defer choosing the exact pattern of recursion to implementation time. Sometimes you don't realize that a pattern won't work until you really start working with the details.

If your goal is to work with a vector in its entirety, then the extra parameters change the basic interface to that problem. To hide them we use a OVERLOADED function where one version takes on the vector and calls the recursive procedure with first equal to zero and last equal to the size of the vector. This will cause the recursive variant of

the function to consider all the elements of the vector.

Figure 1 gives the basic template for first + rest structural recursion on vectors with a top-level variant that works on the whole vector by using the recursive variant as a helper. We'll be applying this template to search and sort.

```
/**
 *
 *@param v the vector
 *@param first the smallest index to be processed
 *@param last the excluded upper bound of the interval of indexes
     to be
 * processed
 *@return ...
 *@pre 0 <= first,last < v.size()
 *@post ...
 */
... foo([const] std::vector< ... >& v, int first, int last){

  if( first >= last ){
    // base case
    ....
  }
  else{
    ... v[first] ... foo(v,first+1,last)
  }

}

/**
 *
 *@param v the vector
 *@return
 *@post ...
 */
... foo([const] std::vector< ... >& v){
  ... foo(v,0,v.size()) ...
}
```

Figure 1: Structural Recursion Template for Vectors

Search: The Problem

Search is a fundamental problem in computing. Given a collection[4], find a specific element, typically called the *search key*[5] in that collection. Several variations can occur: find the first occurrence, the last occurrence, and more generally the $k-th$ occurrence. With the *std::vector*, we want to return the index of occurrence. A simpler

[4] std::vector for now

[5] or just *key*

version of search is the *contains* predicate that returns true if the collection contains at least one occurrence of the search key. Finally, another search-like procedure is a counting procedure that returns the total number of occurrences.

For these notes we'll look at the version of search that examines the entire vector and returns the index of the first occurrence of the key. The declaration for this function is given in figure 2 with tests in figure 3.

Figure 2: The Basic Search Function

```
/**
 * Compute the location of the first occurrence of the integer key
 * in the vector data.
 * @param data vector of integers
 * @param key search value
 * @return -1 if key is not found, otherwise the index where key
 * is first found
 * @pre none
 * @post none
 */
int search(const std::vector<int>& data,int key);
```

Figure 3: Tests for Basic Search

```
TEST(search,all){

  EXPECT_EQ(-1,search(std::vector<int>({}),1));
  EXPECT_EQ(-1,search(std::vector<int>({2}),1));
  EXPECT_EQ(0,search(std::vector<int>({1}),1));
  EXPECT_EQ(1,search(std::vector<int>({1,3,5}),3));
  EXPECT_EQ(0,search(std::vector<int>({1,3,1}),1));
  EXPECT_EQ(2,search(std::vector<int>({1,3,5}),5));

}
```

This basic interface and the tests should work regardless of the underlying implementation technique. How you solve the problem does not change the problem itself. In this case, we're interested in searching an entire vector and this function captures that problem succinctly.

Search: The Solutions

The recursive and iterative versions share a lot in common. We'll start with the recursive variant and then look at the iterative version. In both cases it's clear that we only need read-only access to the

vector and using *pass-by-const-reference* would be a good idea.

Recursive Search

For the recursive implementation we need the variant of the search that accepts the bounds of the search range in order to recurse along that interval as shown in figure 4. It's wroth noting this function, as declared, doesn't need to be recursive. We can, and should, view it as a more general version of search: find the key within this subsection of the vector. The more general problem of searching a part of a vector gives us the flexibility we need to enable recursion.

Figure 4: Recursive-Capable Search

```
/**
 * Compute the location of the first occurence of the integer key
 * in the vector data for the index range [fst,lst).
 * @param data vector of integers
 * @param fst the lower bound of the search range
 * @param lst the excluded upper bound of the search range
 * @param key search value
 * @return -1 if key is not found, otherwise the index where key
 * is first found
 * @pre fst <= lst
 * @post none
 */
int search(const std::vector<int>& data,int fst, int lst, int key);
```

When testing the more generic search we should test it not only for whole vector searches, the problem we originally set out to solve, but for partial vector searches as we see in figure 5. Test the procedure as it stands on its own, not just for some subset of its usage.

Figure 5: Recursive-Capable Search Tests

```
TEST(search,some){

 // search all
 EXPECT_EQ(-1,ln14::search(std::vector<int>({}),0,0,1));
 EXPECT_EQ(1,ln14::search(std::vector<int>({1,2,3,2,1}),0,5,2));
 EXPECT_EQ(-1,ln14::search(std::vector<int>({1,2,3,2,1}),0,5,7));
 // search some
 EXPECT_EQ(3,ln14::search(std::vector<int>({1,2,3,2,1}),2,5,2));
 EXPECT_EQ(-1,ln14::search(std::vector<int>({1,2,3,2,1}),2,3,2));

}
```

The top-level search[6] simply calls the more generic version with the interval for the entire vector as shown in figure 6.

[6] search the whole vector

```
int search(const std::vector<int>& data,int key){
  return search(data,0,data.size(),key);
}
```

Figure 6: Top-Level Search: Recursive Implementation

To work out the recursive implementation of the generalized search we start with the base case. When a vector is empty, it cannot contain the key so return -1. With the non-empty case we work out the problem in terms of the first[7] and the result of recursing on the rest[8]. In the context of search this boils down to the following observation: either the first is the key, the key is in the rest, or the key is not in the vector at all. A complete case analysis reveals four possible situations[9]: the first is the key and the rest contains the key, the first isn't the key and the rest contains the key, the first is the key and the rest doesn't contain the key, or the key is neither the first nor is it in the rest. By recursively calling search on the rest we expect to get the location of key in the rest or a -1 if it's not in the rest. For example, if the vector v contains $\{2, 3, 2, 1\}$ and the search key is 2 then the recursive call *search(v,1,v.size(),2)* should return 2, but because 2 is also at location 0, the first, we should return 0 in favor of 2. If we were searching for the key 4 then the recursive call should return -1 and given that the first, 2 isn't 4, we should return -1. By continuing to work examples like this we see that the recursion is necessary if and only if the first isn't the key. In the case where the first element is the search key, we should just return the first index without recursively searching the rest.

[7] data[fst]

[8] search(data,fst+1,lst,key)

[9] two potential locations (first and rest) with two potential states(contains and doesn't contain)

In figure 7 we see the finished code for the recursive search. The two cases of the non-empty portion of the main conditional have been flattened into the main conditional itself rather than nesting a second conditional in the else of the main conditional.

```
int search(const std::vector<int>& data,int fst, int lst, int key){
  if( fst >= lst ){
    return -1;
  }
  else if( data[fst] == key ){
    return fst;
  }
  else{
    return search(data,fst+1,lst,key);
  }
}
```

Figure 7: Search Some: Recursive Implementation

It's worth stopping to check our implementation against our case analysis. The most important case is the recursive one: the search range of the vector is either empty or it's not. Empty case is handled by the *if* and the non-empty case is caught either by the *else if* or the *else*. When the search range isn't empty we can condense the four cases down to two: the key is the first or its existence and location can be determined through recursively searching the rest. The *else if* case should catch the case where the first is the key. The *else* covers the other case.

Iterative Search

The basic logic of iteration is to traverse the structure and accumulate the solution while you traverse. Vector traversals work the same way as string taversals. You simply count your way through the set of index values[10]. By default we start at zero and count up, but other counting schemes are possible. The solution to this problem is an index value or -1. Developing the recursive solution showed us that we shouldn't need to continue searching once we discover the first occurrence of the search key. In an iterative world this means stopping the traversal and returning the current index. Combine this with the fact that we need to accumulate an index value, we notice that an extra accumulator isn't required, the loop is already accumulating what we need in it's counter. We need to be careful though. Basic accumulator logic dictates that we return the accumluated value when traversal is done. When we don't find the search key the loop's counter will be the size of the vector and we want -1. Similarly, if the vector is empty, the initial counter value is 0 and we need to return -1. The fix here is to recognize that when we find the key, we can return the counter, and if we don't find the key we just return the literal -1.

[10] [0,size)

In figure 8 we see the finished code for the recursive search. The two cases of the non-empty portion of the main conditional have been flattened into the main conditional itself rather than nesting a second conditional in the else of the main conditional.

It's worth your time to really analyze this solution and clearly identify how we tweaked the basic "traverse and accumulate" logic to arrive at this implementation. Imagine we're searching the vector v containing $\{2, 3, 1, 2\}$ for the search key 1. If we're currently looking at index $i = 1$ then we've accumulated the fact that everything before 1 doesn't contain the key[11]. The vector element at 1 isn't the key, so we should leave the accumulated value as -1. If the key were 2 then we should have discovered the key on a previous iteration and the accumulator would be 0, the location of the first occurrence of the key in the previously traversed structure. Finally, consider the

[11] i.e. our accumulator is -1

```
int search(const std::vector<int>& data,int fst, int lst, int key){

   for(unsigned int i{0}; i < data.size() ; ++i){
    if( data[i] == key ){
      return i;
    }
   }

   return -1;
}
```

Figure 8: Search: Iterative Implementation

case where the key is 2 and the current location is 3. We should have discovered 2 at location 0 on a previous iteration and upon rediscovering 2 and location 3 we should preserve the previous accumulator value because our goal is to find the first occurrence. If you wanted to capture this kind of on the nose iterative thinking in code you'd end up with what you see in figure 9. From this perspective our version is an optimization of the full traversal version[12].

[12] why keep going once you've found it and why accumulate the location twice, once for the traversal loop and once for the iteration

```
int search(const std::vector<int>& data,int fst, int lst, int key){

  int loc{-1};
   for(unsigned int i{0}; i < data.size() ; ++i){
    if( data[i] == key && loc == -1 ){
      loc = i;
    }
   }

   return loc;
}
```

Figure 9: Search: Iterative Implementation with explicit accumulation and full traversal

Sort: The Problem

Sorting is something we teach small children before they even learn to read and do arithmetic. The problem is that fundamental to our thinking. It may come as a surprise then that it provides a wide array of variations. For starters, we can sort in ascending or descending order. As a computing problem we can think of it as a function and a mutator. The the case of the former, we'd be producing a second vector with the same contents as the argument but a potentially different order. As a mutator we would be committing to rearranging the contents of an existing vector. Of course, we could implement

a function via mutation of a local vector and we can implement a mutator by making a local sorted copy then swapping the original for the contents of the sorted copy.

The version of sort we'll be looking at is an ascending order, IN-PLACE sort. Working in-place is just another way of saying we want to implement sort as a vector mutator and manage the mutation directly rather than copying the contents and modifying the copy. In-place sorts are important to computing because they guarantee we do not make copies of data and in general, copying can be costly.

Another desirable property of sorts is STABILITY. A stable sort will leave equivalent values in the same relative order as they were originally[13]. We'll set aside stability as a design goal for now but should step back from whatever implementation we end up with and determine if we just so happened to produce a stable sort.

[13] the first 1 is still first, the second 1 is still second, and so on

In figure 10 we see the declaration for a sort mutator. The accompanying tests are give in figure 11. When developing tests it's good to work examples in increasing order of complexity. This typically boils down to increasing sizes and maybe different cases for a particular size. For vectors of size zero or one, the sort seems trivial. There's really nothing to do. For a size of two we can get a feel for when work does and does not need to happen because sometime we need to swap the order and sometimes we don't. Finally, vectors of size greater than run a wide gambit of nearly sorted to totally unsorted.

Figure 10: Sort as a Mutator

```
/**
 * Sort the contents of data in least to greatest order
 * @param data vector of integers
 * @return none
 * @pre none
 * @post contents of data have been sorted in least to greatest
     order
 * for all i in [0,data.size()-1), data[i] <= data[i+1]
 */
void sort(std::vector<int>& data);
```

This time we'll start by looking at the iterative implementation because loops and iteration plan nice with mutation and perhaps we'll have an easier time teasing out basic logic in that environment. Once we've teased out some iterative sorting logic, we'll see if that doesn't provide insight into a recursive strategy.

```
TEST(sort,all){

  std::vector<int> sortme;

  sort(sortme);
   EXPECT_EQ(std::vector<int>({}),
       sortme);

   sortme = std::vector<int>({1});
  sort(sortme);
   EXPECT_EQ(std::vector<int>({1}),
       sortme);

   sortme = std::vector<int>({7,4});
  sort(sortme);
   EXPECT_EQ(std::vector<int>({4,7}),
       sortme);

   std::vector<int> data{8,7,6,5,4,3,2,1};
  sort(data);
   EXPECT_EQ(std::vector<int>({1,2,3,4,5,6,7,8}),
       data);

}
```

Figure 11: Sort as a Mutator

Iterative Implementation

Iterative procedures still have bases cases, cases where no iteration[14] is needed. The question we ask ourselves is: when can we tell that a vector is sorted without actually traversing the vector in any way? The answer is when the vector contains one or fewer items. Here we see an instance of a non-empty base case: Vectors of size one need not be worked on because they are trivially sorted already. As we move to implementing our sort we want to ensure that our traversal loop only traverses vectors with size greater than one.

[14] traverse and accumulate

Iterative mutation is about accumulating effect, not value. If we've done i steps of the sorting traversal than the vector should be sorted in the first i places. This also means that when the loop is done, the sort should be done. There's nothing left to do. We need to combine this mutation based thinking with the base case now. If the base case includes regions of size one, then our traversal should begin at the second element in the vector and accumulate effect across the whole region of size two. This all pushes us to a loop design that begins at $i = 1$ and counts up towards the size of the vector. Beginning at one means the loop will only perform iterations for vectors with a size greater than one[15].

[15] assuming we continue as long as $i < size$ of course

The last thing we need to work out is how to accumulate. Let's look an example. If the vector v originally contained $\{3, 7, 1, 2, 8, 6\}$ and we've performed 2 iterations, then the current index i should be 3 and v from 0 to 2 should contain $\{1, 3, 7\}$. We're now faced with the problem of modifying the region from 0 to 3 such that the 2 at location 3 is placed at location 1 and the 3 and 7 following it are shifted over to the right one. It is not immediately clear how to do this.

Because we're trying to work in-place, it's important that we think in terms of moving data within the existing vector and not using any kind of operation that changes the size of the vector. This rules out the use of the std::vector *insert* and *erase* methods. There is an *assign* method[16] that might be useful, but in order to use it we have to know where, exactly, things need to go. When we look at our example, we see what needs to go where because our visual processing lets us work out the global structures. The computer doesn't see that. It sees nothing. The computer has a sorted region and an element right next to that region that needs to get moved so that the whole region is now sorted. There appears to be no pre-build solution to this[17], so the answer is, of course, to abstract away the problem in the form of a helper.

[16] `http://www.cplusplus.com/reference/vector/vector/assign/`

[17] or we just want to solve this problem on our own anyway

When faced with a task for which no known method, procedure, or operation provides an solution you should quickly design a helper

to solve the problem. This is the bread and butter of top-down design[18]. Find the helpers you need by working them out in context. What we need is a procedure that takes the vector and the bounds of a region of that vector. To shake things up we'll include the upper bound of the region this time. So, our precondition is that for the region contained in $[fst, lst]$, the data in $[fst, lst)$ is sorted. We know nothing about the element at $lst - 1$. The procedure should then move the data at $lst - 1$ such that everything in $[fst, lst)$ is now sorted. We're also assuming that the sorted region contains at least one item. This means $fst < lst$. In short, this procedure inserts the item at *lst* into the region to the right of *lst* and that that ends at *fst* so that the whole region is sorted. We can now properly declare and document this as a C++ procedure as shown in figure 12.

[18] The real win here is that you're able to solve the problem without actually solving the problem by clearly identifying, naming, and specifying the missing piece(s)

Figure 12: Insert: Declaration

```
/**
 * Move data[lst] into sorted region data[fst..lst-1] such that
 * the whole region is sorted.
 * @param data vector of integers
 * @param fst lowest index of region for insertion
 * @param lst the location of the item to be inserted. Also
 * one more than the last of the insertion region
 * @pre fst < lst. data[fst .. lst-1] is sorted in
 * least to greatest order
 * @post data[fst..lst] is sorted in least to greatest order
 */
void insert(std::vector<int>& data,
          unsigned int fst, unsigned int lst);
```

A few tests for insert are given in figure 13. Once again, develop tests ranging from simple base cases to larger more complex cases. We should also test insert in a more general setting. We expect the value of *fst* to always be 0 but it doesn't need to be. Testing more general cases lets you re-evaluate and reaffirm the pre and post conditions.

Now that it's clear what *insert* should do and how to use it, we can finish up *sort*. In figure 14 we see the complete iterative sort known as Insertion Sort. Insertion sort is a class, well known algorithm for sorting algorithm. Our design uses an insert helper. It's not uncommon to see the helper replaced by the actual insertion logic. This design clearly separates the core iterative logic of the sort from the accumulative operation carried out by insert. We're also able to express *sort* without knowing how to insert. In short, this design clearly separates different concerns of the problem when compared to the explicit, all-in-one version you'll find in texts and

```
TEST(insert,all){

   std::vector<int> testme({3,2});
   ln14::iter::insert(testme,0,1);
   EXPECT_EQ(std::vector<int>({2,3}),
       testme);

   testme = std::vector<int>({2,3});
   ln14::iter::insert(testme,0,1);
   EXPECT_EQ(std::vector<int>({2,3}),
       testme);

   testme = std::vector<int>({2,4,5,7,2,4});
   ln14::iter::insert(testme,0,4);
   EXPECT_EQ(std::vector<int>({2,2,4,5,7,4}),testme);

   testme = std::vector<int>({9,4,5,7,2,1});
   ln14::iter::insert(testme,1,4);
   EXPECT_EQ(std::vector<int>({9,2,4,5,7,1}),testme);

}
```

Figure 13: Insert: Tests (Iterative Version)

online.

```
void iter::sort(std::vector<int>& data){

  for(unsigned int i{1}; i < data.size(); i++){
    iter::insert(data,0,i);
  }
  return;
}
```

Figure 14: Sort: Iterative Sorting

We now need to finish the implementation of *insert* if we want sort to actual work. If our sort is to be an in-place mutator, then *insert* must also work in-place through mutation. Once again, we'll choose to use iteration rather than recursion in order to implement insert.

The first thing we need to recognize is that *insert* needs to traverse and accumulate through the portion of the vector indexed by $[fst, lst)$. The in-place requirement really pushes us away from the standard fst to lst traversal though. Let's see why. Say some vector v contains $\{1,3,5,7,2\}$ in the $[fst, lst]$ range[19]. After a single iteration from fst to lst, what should the vector range look like? What we need to remember is that the region we've traversed should be the final solution with respect to that portion of the vector. So after a single

[19] So v[fst] is 1 and v[lst] is 2

iteration the fst should still be 1. After two iterations we should see $\{1,2,3,5,7\}$ because the 2 needs to be at $fst+1$, the second spot, in the final solution. If we're to do this in place, then we need to shift everything in $[i, lst)$ to the right one so that $v[lst]$ can go at location i. Once again, it seems like we'd need a helper for this because there's no operation for shifting a region. Before we go down that path, let's explore a different traversal patter: lst-1 to fst.

When traversing starting at lst-1 and working down towards fst we need not only rethink the loop counting pattern but the iterative logic. The region we've traversed already is not at the back of the range, not the front. After one iteration with our same example vector v, what should v look like? The basic idea now is that if we're dealing with location i on the current iteration that after this iteration $[i, lst]$ should be sorted and contain all the values that were originally in $[i, lst]$. So after one iteration our vector should look like $\{1,3,5,2,7\}$. What we've actually done is reversed the problem a bit. We want to insert the value at lst into the sorted region $[fst, lst)$. In working lst to fst, we actually inserting the last of $[fst, i]$ into the sorted region $(i, lst]$. Initially the region $(i, lst]$ is just lst. After the first iteration, it's $[lst-1, lst]$, and then on down.

The fact that both regions are sorted lets us cut some corners. Let's trace insert a bit to find it.After one iteration our example region of v contains $\{1,3,5,2,7\}$. We can accomplish this by swapping the last two elements or more generally $v[i] and v[i+1]$. On the second iteration we need to "insert 5 into $\{2,7\}$". Once again a simple swap would accomplish this. In fact, if you continue on you'll find that all we ever need to do is swap $v[i]$ with $v[i+1]$ until the original insertion number, 2, gets to its final location at which point we can just stop traversing all together. Why is this? Until we get the original $v[lst]$ to the correct spot, $v[i+1]$ is always that number and everything to its right (if it exists) must be greater than or equal to all the stuff in $[fst, i]$ because that makes up the original sorted region $[fst, lst)$! When we swap $v[i]$ with $v[i+1]$ we're simply putting a number back where it was but shift to the right of the original $v[lst]$ rather than the left of $v[lst]$ where it started. When $v[i]$ is less than $v[i+1]$, then we can stop the traversal because we know everything to the left of $v[i]$ (if it exists) is less than or equal to $v[i]$ and must therefore be less than $v[i+1]$. By traversing from right to left rather then left to right reversed the problem a bit but in doing so naturally solved our shift and place problem by turning the shift into iterated swaps.

The final version of iterative *insert* is shown in figure 15. It's design and implementation merit repeated study as it illustrates how varying the traversal pattern can sometimes avoid some problems

encountered with the standard left to right traversal at the cost of forces us to reverse our thinking on the iterative logic of the problem. The other thing you'll see is careful management of counting down through an index range using unsigned integers. If fst is 0[20], then when the unsigned int i is equal to fst and we do $i--$ we won't get -1 but a very large number instead. The check against the size of the vector ensures we never carry out an iteration with an i value outside of $[fst, lst]$.

[20] and it always will be for our target usage of insert

Figure 15: Insert: Iterative Implementation

```
void iter::insert(std::vector<int>& data,
          unsigned int fst, unsigned int lst){

  for(unsigned int i{lst-1}; i >= fst && i < data.size(); i--){
    if( data[i+1] < data[i] ){
      std::swap(data[i],data[i+1]);
      }
      else{
     return;
      }
   }
   return;
}
```

The important take away here is that insertion sort is what you'd eventually discover if you apply basic iterative design with respect to the structure of the vector. In that sense it is the most natural sort from a basic computational perspective.

Recursive Implementation

If insertion sort is what comes from basic iterative design, then what comes from basic recursive design? What we've learned about sort generally is that any vector of size one or smaller is trivially sorted. This is our recursive base case and it's worth noting that it boils down to two cases[21] as opposed to a single, usually empty, non-recursive base case. This is OK! The only hard and fast rule is that you identify at least one non-recursive case and that your recursive case eventually decomposes to one of those cases.

[21] empty and size 1

Applying the basic recursion strategy for sort means recursively sorting the rest then combining the first with that in some way such that we achieve the desired result of a completely sorted vector. As we learned from searching, working with vectors recursively requires more general versions of our functions that allow us to set arbitrary ranges of the vector to be worked with. For sort that means the function give in figure 16.

```
/**
 * Sort the contents of data in least to greatest order
 * @param data vector of integers
 * @param fst the lower bound of the sort range
 * @param lst the excluded upper bound of the sort range
 * @return none
 * @pre fst <= lst
 * @post contents of data[fst..lst-1] have been sorted in least to
     greatest order
 * for all i in [fst,lst-1), data[i] <= data[i+1]
 */
void sort(std::vector<int>& data,int fst, int lst);
```

Figure 16: Sort: Generalized version that supports Recursion

This function should behave as shown in its tests give in figure 17. Once again, it's good to treat this function as a standalone entity and write tests for cases we don't intend to use, namely sorting interior portions of the vector.

We can now complete the top-level sort as a special case of the more general sort procedure as shown in figure 18.

With the main, top-level procedure done it's time to work out the recursive logic for the helper. The base case occurs if size is less than or equal to one. This means that *fst* is greater than or equal to *lst*. When this occurs, we simply return and leave the vector untouched. When the portion of the vector we wish to sort contains more than one element, then we need to sort. We do so by first sorting the rest, namely $[fst + 1, lst)$. If the portion of the vector we wish to sort originally contains $\{3, 6, 1, 4, 8\}$, then the recursive call to sort should leave us with $\{3, 1, 4, 6, 8\}$. The final "operation" must move the 3 in the *fst* position between the 1 and 4. Once again, what we really need here is an *insert* operation. This time we need to insert an element immediately to the left of the sorted region where in the iterative case the element was to the right. The documentation and declaration of our new insert is in figure 19 with tests found in figure 20.

It's important to stop for a second and think about why it is that we need a different insert. What would have happened if we recursively sorted all but the last of the vector region containing $\{3, 6, 1, 4, 8\}$? This would have left us with $\{1, 3, 4, 6, 8\}$[22]. We'd still need insert and this time we'd need the exact same variation of insert that we used in our iterative solution. For practice, we'll continue forward with our new insert and this time implement it recursively as well. If we wanted to, we could recurse on all but the last and then just use our iteratively implemented insert to finish the job.

[22] which is fully sorted by coincidence only

The base case of a recursive insert occurs when the vector region

```
TEST(sortrecur,some){
   std::vector<int> sortme;

   sort(sortme,0,sortme.size());
   EXPECT_EQ(std::vector<int>({}),sortme);

   sortme = std::vector<int>({5});
   sort(sortme,0,sortme.size());
   EXPECT_EQ(std::vector<int>({5}),sortme);

   sortme = std::vector<int>({5,7});
   sort(sortme,0,sortme.size());
   EXPECT_EQ(std::vector<int>({5,7}),sortme);

   sortme = std::vector<int>({7,5});
   sort(sortme,0,sortme.size());
   EXPECT_EQ(std::vector<int>({5,7}),sortme);

   sortme = std::vector<int>({5,1,4,3});
   sort(sortme,0,sortme.size());
   EXPECT_EQ(std::vector<int>({1,3,4,5}),sortme);

   sortme = std::vector<int>({5,1,4,3});
   sort(sortme,2,sortme.size());
   EXPECT_EQ(std::vector<int>({5,1,3,4}),sortme);

   sortme = std::vector<int>({5,1,4,3});
   sort(sortme,1,sortme.size()-1);
   EXPECT_EQ(std::vector<int>({5,1,4,3}),sortme);
}
```

Figure 17: Sort: Tests for Generalized version that supports Recursion

```
void sort(std::vector<int>& data){
   sort(data,0,data.size());
   return;
}
```

Figure 18: Sort: Top-Level Recursive Procedure

```
/**
 * Move data[fst] into sorted region data[fst+1..lst] such that
 * the whole region, data[fst..lst] is now sorted.
 * @param data vector of integers
 * @param fst lowest index of region for insertion, location to be
     inserted
 * @param lst the upper bound of the sorted region
 * @pre fst < lst. data[fst+1 .. lst-1] is sorted in
 *  least to greatest order
 * @post data[fst..lst] is sorted in least to greatest order
 */
void insert(std::vector<int>& data, unsigned int fst, unsigned int
    lst);
```

Figure 19: Insert: Another variation of Insert

```
TEST(insertrecur,all){
   std::vector<int> actual;

   actual = std::vector<int>({1,2});
   insert(actual,0,actual.size()-1);
   EXPECT_EQ(std::vector<int>({1,2}),actual);

   actual = std::vector<int>({2,1});
   insert(actual,0,actual.size()-1);
   EXPECT_EQ(std::vector<int>({1,2}),actual);

   actual = std::vector<int>({5,2,4,6});
  insert(actual,0,actual.size()-1);
   EXPECT_EQ(std::vector<int>({2,4,5,6}),actual);

   actual = std::vector<int>({3,1,5,2,4,6,1});
  insert(actual,2,5);
   EXPECT_EQ(std::vector<int>({3,1,2,4,5,6,1}),actual);
}
```

Figure 20: Insert: Tests for the new Insert

is empty and all we're looking at is the element we wish to insert. When this happens we simply return and leave the vector untouched. From our work with an iterative insert we learned that the strategy that seems to work for the recursive case is to repeated swap the target element with adjacent elements from the sorted rejoin[23] seems to get the job done. We can employ this same strategy using a recursive procedure. If the target element, the one at *fst* is greater than the one next to it, then swap. We then recursively insert on $[fst + 1, lst]$ with the item to be inserted now stored at $fst + 1$ and the old $fst + 1$ at fst. If the target element isn't greater than the first of the insertion region, then it's in the right place and we can return without swapping or recursively inserting any more. .

[23] or conversely swapping the next element in the sorted region with the target element

Our recursive insertion logic is written up in C++ in the insert implementation given in figure 21.

```
1  void recur::insert(std::vector<int>& data,
2             unsigned int fst, unsigned int lst){
3    if( fst >= lst ){
4      return;
5    }
6
7    if( data[fst] > data[fst+1] ){
8      std::swap(data[fst],data[fst+1]);
9      insert(data,fst+1,lst);
10   }
11   return;
12 }
```

Figure 21: Insert: A recursive implementation

Structure-Oriented Thinking

The single most important observation to make in all this is that we're able to solve problems using iteration and recursion by reasons through the structure of the data. When we recursively process the rest, we do so with the values in that part of the structure. Similarly, the first *i* elements that we've iterated over in an iterative solution were the values that were in that part of the vector to begin with. This means we can tease out solutions to problems without worrying too much about the specific values encountered. It also highlights the importance of identifying and understanding structural patterns within your data as these patterns give you an in to solving problems with and about that data.

Once we understand the nature of structural thinking, it's clear that recursion and iteration, when done based on structure, have

an awful lot in common. They both work off the same underlying principles, the structure of data, but do so through different means. For that reason it is unsurprising that both ways of thinking led to similar, if not logically equivalent solutions to our problems.

The natural scientific question here is: what are the limits on these structural strategies? On one hand this is a question of COMPUTABILITY: are there problems for which structural thinking will not yield a solution? On the other hand this a question of COMPLEXITY: are the procedures produced by structural thinking optimal or can we do better? Setting aside the big picture questions, we might simply want some rigorous means of comparing iterative and recursive solutions to one another and comparing structural design to other strategies more generally. To do this we'll turn to questions of complexity. While computability seems important, time has shown that there seems to be no shortage of interesting problems that computers can solve and that more often than not, it's the time and resources needed to solve those problems that pose challenges.