COMP 220
Lecture Notes 02
Structural Recursion and Iteration

August 25, 2016

> Like most data structures texts, chapter 10 of our book introduces you to several sorting[1] algorithms and discusses the complexity of each algorithm. The goal of these notes is to look not at the algorithm by the underlying design principle that leads to the algorithm. We'll then use the resultant algorithms to examine general limitations on the complexity of that design principle.

[1] Selection Sort, Mergesort, and Quicksort

Structural Design

Our first design principle yields linear search, and insertion sort. It can be carried out either iteratively or recursively. The core idea is:

> The structure of the logic of your algorithm mirrors the structure of your data.

Structural design first begins with identifying the underlying structure of your data and then using that to bootstrap the algorithm. You first encountered this with list processing functions in Racket. Lists have a basic recursive structure. They come in two cases: empty or not. The non-empty list is composed of a singular datum, the first, and a sub-list, the rest. Because the list is inherently recursive, structural design dictates that our list algorithms should also be recursive. In a C++ context, we'd have the following procedure template:

```
1  ... functionName(recursiveType alst, ... ){
2    if( isEmpty(alst) ){
3      ...
4    }
5    else{
6     ... first(alst) ...
7     ... functionName(alst, ...) ...
8    }
9  }
```

Figure 1: The basic template for a structurally recursive procedure

Notice the requirements for applying the template given in figure 1 are that the type of *alst* provides a means of distinguishing the empty case from the non-empty case, a means of selecting the first element, and a means of selecting the rest. All of these operations occur in $O(1)$ time and are therefore not an impediment to designing efficient algorithms based on this pattern.

When we encountered vectors in C++ we discovered a data structure based on easily random access via numerical indexes. The trade-off for random access in vectors is that recursive decomposition is either not supported or inefficient[2]. Rather than work with structure recursively, we used an iterative traversal of the index range. By counting through the indexes we can access each element of the vector in turn. This gives way to the state-driven "traverse and accumulate" pattern that typifies iterative algorithms.

[2] $O(1)$ in this case

```
1  ... functionName(recursiveType avec, ... ){
2
3    ... accumulator{...};
4    for(int i{0} ; i < size(avec) ; ++i ){
5      accumulator = ... accumulator ... avec[i] ...
6    }
7    ... accumulator ...
8  }
```

Figure 2: The basic template for a structurally iterative procedure

Variations on a Theme

At this point you should be familiar with the patterns hinted at in figure 1 and 2. The key moving forward is to recognize that structural design is not only these two patterns but whole class of patterns typified by these two.

Recursion doesn't require an empty base case, just at least one base case. Your base case can be a singleton element or any fixed, finite number of elements. You can have more than one base case. What's important is that you have *at least one case in which your structure and your algorithm does not reference itself*. The recursive, non-empty case doesn't need to be the first and the rest. It can be the last and all but the last. It can be the first two and everything after that[3]. What's important is that deconstruction by parts is efficient[4] and that *recursively applying the deconstruction will eventually converge to the base case*.

[3] notice this pairs well with a size 2 base case!
[4] $O(1)$ ideally

Iteration doesn't have to occur on a left to right basis. You can count through the indexes from last to first, odds then evens, or some other counting pattern. The important thing is that *your traversal pattern visits all the relevant elements of the structure*. Accumulation isn't always explicit. It's sometimes possible to implicitly accumulate and avoid updates to the accumulator that do not change its state.

Often we can induce recursive or iterative structure where it is not the default. We can treat the index range of a vector as a recursive structure and recursive over an index counter. Similarly, it's possible

to iterate on a list by keeping a state variable that tracks the current location. When tweaking new structures out of our data we simply need to ensure that the operations which enable the structural recursion and iteration are efficient.

Ultimately, what makes a strategy a structural one is that you are attempting to map the logic of your algorithm on to a basic structure within the data. The strength of this approach is that it is value agnostic. You don't concern yourself with the values within the data structure, you just concern yourself with how data is organized within the structure itself.

Structural Search and Sort

Linear search is what you get if you apply structural design to the problem of search. Insertion Sort is what you get when you apply the design priciple to the problem of sorting. These algorithms were covered and analyzed in Lecture Notes 16 from COMP161[5] so we'll review them briefly here and reiterate how structural design gives rise to these algorithms.

[5] https://jlmayfield.github.io/MC-COMP161/

Search

Iterative search traverses the vector indexes and accumulates the current index of the search key using -1 as a indicator for "not found". When searching for the first occurrence of a key or the mere presence of a key we can forgo the explicit accumulation and return the index of the key within the loop when it's found or -1 if the loop terminates because no occurrence of the key was found. Thus, this formulation of the algorithm constitutes an optimization of the basic structural version.

```
1  int search(const std::vector<int>& data,int fst, int lst, int key){
2
3     for(unsigned int i{0}; i < data.size() ; ++i){
4      if( data[i] == key ){
5        return i;
6       }
7     }
8
9     return -1;
10 }
```

Figure 3: Search: Iterative Implementation

A vitally important way of thinking about structural recursion is to imagine the process at some time step t given that there are more

than t elements in the vector. This divides the vector into three regions. Elements from 0 to $t-1$ are those we've searched through already. The element at t is the element we are currently working with. Finally, the elements from $t+1$ to the end are those which we have yet to traverse. The item you're looking for is either not in the vector or it's in one of these three regions. If it is the first region, those we've previously traversed, then we should have found it already[6]. If it's in the region we have yet to traverse, the third region, then assume you'll find it later on. These assertions are what as known as **loop invariants** and can act as the **inductive hypothesis** in a proof of the correctness of the loop and algorithm generally. As you get deeper into discrete mathematics you'll learn [7]. If you want to prove a structurally oriented algorithm correct, then odds are you'll use mathematical induction. The keystone logic is relative to the element at t. We, the programmer must come up with the logic that determine if the element at t is the key. If we can do that and we can properly traverse all of the vector, then our algorithm will work.

[6] so either we've returned that index or we've accumulated it's value

[7] proof by induction

To search the vector recursively we need to leverage the recursive structure of the index range and use a helper procedure that tracks the index of the first and last[8]. The helper is technically a more general implementation of search that allows searching the region of the vector indexed by $[fst, lst)$.

[8] technically we don't need the track the last, but doing so opens up alternate patterns of recursion

Figure 4: Search: Recursive Implementation

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

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);
  }
}
```

Once again, the underlying logic is that of induction. We assert that if the key is in the rest of the vector, i.e. $[fst+1, lst)$, and we recursively search the rest of the vector, then that recursive call to search returns the index of the key. We then focus on the logic of examining the first element. The logic shown in figure 4 optimizes on this by recognizing that if you need the first occurrence, then it's

better to check the first and return if it's the key prior to making the recursive call.

Sort

Structural sorting starts in the same fashion as searching. The problem we face is that the operation we need to repeat, i.e. what we do to the first when recursing or what we do to the i^{th} when iterating, is complex enough as to merit it's own separate design consideration. With search, what you do to the single element is simple enough that we can just do it. With sort, it's not initial obvious and falls clearly into the camp of, "design a helper". In both cases the logic of the helper is that of insert.

In the case of the iterative insertion sort we assume that at step t, the elements at 0 to $t-1$ are sorted and we must incorporate the t^{th} element such that all $t+1$ elements up to and including at t are sorted after the t^{th} iteration. This is an inherently structural argument because we're thinking in terms of the index range and vector structure and not about the values in the vector per se.

Insert itself can be solved structurally. The implementation shown in figure **??** traverses down from last to first and swaps adjacent elements as it goes. It's optimized by stopping before the first when it's determined that no swap is needed[9].

[9] this means it's all sorted

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

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

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;
}
\label{code:isortiter}
\caption{Sort \& Insert: Iterative Implementations}
```

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

void recur::sort(std::vector<int>& data,int fst, int lst){
  if( fst >= lst-1 ){
    return;
   }

   recur::sort(data,fst+1,lst);
   recur::insert(data,fst,lst-1);
   return;
}

void recur::insert(std::vector<int>& data,
        unsigned int fst, unsigned int lst){
   if( fst >= lst ){
     return;
   }

   if( data[fst] > data[fst+1] ){
     std::swap(data[fst],data[fst+1]);
     recur::insert(data,fst+1,lst);
   }

}
```

Figure 5: Sort & Insert: Recursive Implementations