COMP220
Lab 4 & Homework 4

Solutions

Fall 2017

Abstract

For this lab you'll trace and illustrate the action of the key sorting and searching algorithms that we've been studying in class. Code for these algorithms can be found on the server in the course home directory (`/home/comp220/f17`, in the `lab4` subdirectory). You should follow the code line for line and be certain that you understand how it implements the high level logic discussed in class. Your goal for the lab period is to make it through the helpers. We'll continue working on the sheet in class Friday. The sheet should be completed by class time Monday as a homework assignment.

On Tracing Code

The key to stepping through code is tracking and tracing the STATE of the program. This means keeping tabs on the current argument values for recursive functions, the current value of local variables, and the current value/state of variables being mutated. The ultimate goal is to keep tabs on variable that are mutating and as needed the variables that dictate that mutation. If a loop is mutating a vector, then we track the vector mutation and the value of any loop counters that drive that mutation. At a micro level, you track state from one program statement to the next statement. At a more macro level you track it across core, repeated code blocks. When using iterative loops, this means one iteration to the next. For recursive procedures you track the state and function arguments for each recursive call. In doing this we get a clear picture of how the data changes across the whole of the procedure's execution and, when done right, you get a clear visual history of the process in terms of repeated work.

An excellent way of recording state change is as a table where the process progresses from row 1 down the table and each column corresponds to a piece of state that you're tracking. This gives you a visual record of each major step in the program's execution. You can check yourself by seeing if the number of rows makes sense with what you know about the amount of repetition carried out by the code.

For complex procedures that utilize non-trivial helper procedures, it's helpful to first trace and understand the helpers outside of the context of their repeated usage. You can then treat them as black boxes for the procedure they help. The ultimate goal is to separate the trace of the helper from the trace of the procedure calling the helper. For example, if we know what insert will do, then when tracing insertion sort, we can skip the work done inside insert and worry only about the bigger picture of the sort itself. This means tracking the $O(n)$ calls to insert while not tracking the $O(n)$ steps within the insert itself.

Here we see a trace of recursive insertion sort. In this trace we list both calls to sort and insert to ensure that we see how the two work together in this context. For sort we're listing the state of the vector at the time of calling, for insert we list the state of the vector after insert is complete to help illustrate the nature of the non-tail position recursion used by the algorithm.

```
1  std::vector<int> v{3,1,6,7,2,9,0};
2  structural::recur::sort(v);
```

data	fst	lst	procedure
{3,1,6,7,2,9,0}	0	7	sort
{3,1,6,7,2,9,0}	1	7	sort
{3,1,6,7,2,9,0}	2	7	sort
{3,1,6,7,2,9,0}	3	7	sort
{3,1,6,7,2,9,0}	4	7	sort
{3,1,6,7,2,9,0}	5	7	sort
{3,1,6,7,2,9,0}	6	7	sort
{3,1,6,7,2,0,9}	5	7	insert
{3,1,6,7,0,2,9}	4	7	insert
{3,1,6,0,2,7,9}	3	7	insert
{3,1,0,2,6,7,9}	2	7	insert
{3,0,1,2,6,7,9}	1	7	insert
{0,1,2,3,6,7,9}	0	7	insert

Figure 1: Tracing Insertion Sort by tracking state

The Helpers

We begin by stepping through the major helper procedures utilized across the search and sort algorithms.

1. Iterative Insert
   Trace the action of the following calls to insert. Track the state of the vector after each iteration and the value of i used on the corresponding iteration. Next to each vector state write the value of i corresponding to that iteration.

```
1  std::vector<int> v{1,3,5,2,4};
2  structural::iter::insert(v,0,3);
3
4  std::vector<int> v{7,2,5,6,1};
5  structural::iter::insert(v,1,v.size()-1);
```

   Finally, identify the base case (smallest possible vector input) of this procedure and comment on how this case is handled by this implementation.

   Solution: The first call to insert terminates when no swap is needed for $i = 0$.

data	i
{1,3,2,5,4}	2
{1,2,3,5,4}	1
{1,2,3,5,4}	0

   The second call to insert terminates when no swap is needed for $i = 1$.

data	i
{7,2,5,1,6}	3
{7,2,1,5,6}	2
{7,1,2,5,6}	1

   Insert's base case occurs when the vector size is 1 or 0. This causes a violation of the loop's continuation condition, so the procedure returns without doing anything else.

2. Recursive Insert

Trace the action of the following calls to insert. This procedure is tail recursive, all the work performed locally by a given call to insert is done prior to making the next recursive call. This allows us to effectively ignore the return to the caller from the callee as it has no effect on how the procedure behaves. For each example clearly show the state of the vector prior to making a recursive call, i.e. after all the localized work is done, and track the values of the procedure arguments *fst* and *lst* from one recursive call to the next.

```
1  std::vector<int> v{5,1,3,4,9};
2  structural::recur::insert(v,0,v.size()-1);
3
4  std::vector<int> v{7,4,3,5,9};
5  structural::recur::insert(v,1,3);
```

Finally, identify the base case (smallest possible vector input) of this procedure and comment on how this case is handled by this implementation.

Solution:

data	fst	lst
{1,5,3,4,9}	0	4
{1,3,5,4,9}	1	4
{1,3,4,5,9}	2	4
{1,3,4,5,9}	3	4

data	fst	lst
{7,3,4,5,9}	1	3
{7,3,4,5,9}	2	3

The base case for the recursive insert is the same as for the iterative implementation: inserting with a region of size 1 or 0. The recursive implementation catches the base case and returns without doing anything else.

3. Partition

For the following calls to partition, trace the state of the vector and the value of *rfst* at the completion of the iteration as well as the the value of i used on that iteration. Then, give the value assigned to *pIdx* at the end of the procedure as well as the final state of the vector.

```
1  std::vector<int> v{5,6,1,2,9};
2  int idx{-1};
3  generative::partition(v,0,v.size(),idx);
4
5  std::vector<int> v{4,2,6,3,7};
6  int idx{-1};
7  generative::partition(v,0,v.size(),idx);
```

Once you've traced the above instances of partition, identify the base case for partition and comment on how this implementation handles this case. Finally, if you were given the initial vector and asked to produce the final state after partitioning, how could you place the values in the lower and upper regions without actually tracing the algorithm? Put another way, how can use use this as a black box, data in data out, and still accurately represent the action of this particular implementation?

Solution:

data	i	rfst	pIdx
{5,6,1,2,9}	1	1	
{5,1,6,2,9}	2	2	
{5,1,2,6,9}	3	3	
{5,1,2,6,9}	4	3	
{2,1,5,6,9}			2

data	i	rfst	pIdx
{4,2,6,3,7}	1	2	
{4,2,6,3,7}	2	2	
{4,2,3,6,7}	3	3	
{4,2,3,6,7}	4	3	
{3,2,4,6,7}			2

The case case for partition is vector regions of size one. An if statement catches this and sets the pIdx parameter to first. This algorithm puts the "pivot" (initially in the first spot) in the "middle", with everything to the left smaller than the pivot and everything to the right smaller than it. This may be a little easier to grasp by imagining a temporary vector. First, scan through the vector from `first+1` to `last` and put everything smaller than the pivot into the temporary vector in the order you see the elements, and cross off these elements from the original vector. Then place the pivot in the next spot in the temporary vector. Finally, scan through the original vector and put all the non-crossed-off elements (which must be greater than the pivot) into the temporary array. Our partition implementation puts our vector in the same order as this temporary vector.

4. Iterative Merge

We'll only trace the action of the iterative implementation of merge. It's recommended that you do the recursive version on your own. Here we want to trace the state of the vector into which the data is being merged. This work is carried out across three loops but these loops are all dependent on the same set of local variables and be treated as a single iterative process. You should therefore track the state of the merge vector *temp* after the iteration completes as well as the value of *lcurr*, *rcurr*, and *tcurr* used during the iteration itself. Clearly indicate on your trace where the procedure has moved from one loop to the next.

```
1 std::vector<int> v{1,5,8,2,3,9};
2 structuralDandC::iter::merge(v,0,3,v.size());
3
4 std::vector<int> v{1,2,5,8,2,4,9,10};
5 structuralDandC::iter::merge(v,2,4,6);
```

Finally, merge has several base cases. Identify them and comment on how the procedure handles them.

Solution:

temp	lcurr	rcurr	tcurr	loop
{1,0,0,0,0,0}	0	3	0	1
{1,2,0,0,0,0}	1	3	1	1
{1,2,3,0,0,0}	1	4	2	1
{1,2,3,5,0,0}	1	5	3	1
{1,2,3,5,8,0}	2	5	4	1
{1,2,3,5,8,9}	3	5	5	3

temp	lcurr	rcurr	tcurr	loop
{2,0,0,0}	2	4	0	1
{2,4,0,0}	2	5	1	1
{2,4,5,0}	2	6	2	2
{2,4,5,8}	3	6	3	2

The base cases occur if one or both of the regions are empty. The procedure checks for this explicitly and returns immediately if it's detected.

Divide and Conquer Searching

We now move to tracing simple divide and conquer algorithms. We'll only look at the recursive implementations of these algorithms.

1. Recursive Binary Search

   Binary search is a pure function. No state is mutated by the procedure. All we need to trace here is the value of the parameters *first* and *last* for the recursive procedure and the locally computed *mid* for each recursive call. Your examples are given in terms of the top level function call. You can start your trace with the first call to the recursive helper.

```
std::vector<int> v{1,3,5,7,9,11,13};
structuralDandC::recur::binsearch(v,8);

structuralDandC::recur::binsearch(v,-4);

structuralDandC::recur::binsearch(v,5);

structuralDandC::recur::binsearch(v,13);
```

   Once again, identify the base case and comment on how it is handled by the procedure.

   Solution:

fst	lst	mid
0	7	3
4	7	5
4	5	4
4	4	

fst	lst	mid
0	7	3
0	3	1
0	1	0
0	0	

fst	lst	mid
0	7	3
0	3	1
2	3	2

fst	lst	mid
0	7	3
4	7	5
6	7	6

   The base case occurs when the region left to search is empty, i.e. `first` $\geq$ `last`. The procedure checks for this and returns -1, since this means the key is not in the vector.

2. Recursive kth smallest (Optional)

Note: we did not cover this in class and we're not going to. However, if you'd like 5 bonus points applied to your exam 1 score, complete this trace and then describe in your own words what the algorithm does.

Tracing the kth smallest algorithm is complicated by the fact that it is a randomized algorithm. To get around this we simply choose "random" pivots ahead of time or as we go. For this exercise you'll be given a set of tables of randomly generated, uniformly distributed integers where each table ranges from 0 to some upper bound n for all n from 1 to 19. (These numbers are in the file `randomnumbers.txt`.)If you need a random number from $[a, b-1]$, then you simply find the table for $[0, b-a-1]$, pick the next number on that table, and add a to it. For example, if you need a random number between 5 and 13 including both of those bounds, then you first go to the table for $[0, 8]$, then pick a number, and add 5 to it. Be sure to convince yourself that you understand why this process works. Pull numbers from the tables in order and don't reuse a number. This ensures that we all see the exact same process unfold.

To trace the kth smallest algorithm you should show the state of the vector being partitioned after every call to partition along with the current values for k, *first*, *last*, and the local variable *pidx*. Once again, this is a tail recursive procedure so you should track all the local state as it sits just prior to making a recursive call. The examples begin at the top level, you can start your trace with the first call to the recursive helper function.

```
1 std::vector<int> v{3,4,9,2,1,8,10};
2
3 generativeDandC::recur::kthsmallest(v,2);
4 generativeDandC::recur::kthsmallest(v,v.size()/2);
5 generativeDandC::recur::kthsmallest(v,v.size()-2);
```

Solution: This algorithm returns the kth ordinal, i.e. the kth smallest number. For example, when $k = 1$ this returns the minimum of the vector, while if $k =$ `v.size()` $- 1$ this returns the maximum. Setting $k =$ `v.size()/2` returns the median.

v	k	first	last	pidx	rand#
{1,2,3,4,9,8,10}	2	0	7	2	0
{1,2,3,4,9,8,10}	2	0	2	0	0
{1,2,3,4,9,8,10}	1	1	2	1	0

v	k	first	last	pidx	rand#
{1,2,3,4,9,8,10}	3	0	7	2	0

v	k	first	last	pidx	rand#
{3,4,2,1,8,9,10}	5	0	7	3	5

1 Mergesort & Quicksort

1. Recursive Mergesort
   While tracing mergesort you should treat calls to *merge* as blackboxes. We will not trace them explicitly and simply concern ourselves with their effect relative to the current context. The real trick in tracing mergesort is following the recursion. The algorithm is not tail recursive so the trace must account for what happens when a recursive call returns. Just like with the insertion sort example, your trace should clearly layout the order in which calls to mergesort are actually carried out by the computer. For each call to mergesort track the value of *first* and *last* only. Interleave with your trace of mergesort the calls to the helper procedure merge. For merge track the values of *first* and *last* and the state of the vector after merge returns. Write one procedure call per line and clearly label the line as mergesort or merge. The end result should tell you the exact order in which things are merged and how the recursive calls to mergesort enable this.

```
1
2 std::vector<int> v{5,2,6,3,1,0,4,8,7};
3 structural::recur::mergesort(v);
```

Solution:

data	first	first2	last	procedure
	0		9	sort
	0		4	sort
	0		2	sort
	0		1	sort
	1		2	sort
{2,5,6,3,1,0,4,8,7}	0	1	2	merge
	2		4	sort
	2		3	sort
	3		4	sort
{2,5,3,6,1,0,4,8,7}	2	3	4	merge
{2,3,5,6,1,0,4,8,7}	0	2	4	merge
	4		9	sort
	4		6	sort
	4		5	sort
	5		6	sort
{2,3,5,6,0,1,4,8,7}	0	2	4	merge
	6		9	sort
	6		7	sort
	7		9	sort
	7		8	sort
	8		9	sort
{2,3,5,6,0,1,4,7,8}	7	8	9	merge
{2,3,5,6,0,1,4,7,8}	6	7	9	merge
{2,3,5,6,0,1,4,7,8}	4	6	9	merge
{0,1,2,3,4,5,6,7,8}	0	4	9	merge

2. Quicksort

   Quicksort is a bit easier to trace because it's tail recursive. It still has the difficulty of properly unrolling the sequencing of the two recursive calls and accounting for the randomized pivot selection.

   If you traced the kth smallest algorithm, you may use that technique for accounting for the randomness, restarting your random integer selection from the start of your tables.

   If you did not trace the kth smallest algorithm, think of the call to `setRandPivot` as always choosing `first` as the pivot.

   We'll interleave calls to quicksort and partition. For partition you should track the values of *first* and *last* and record the final state of the vector and *pIdx*. For quicksort just track *first* and *last*.

```
1
2 std::vector<int> v{5,2,6,3,1,0,4,8,7};
3 generative::recur::quicksort(v);
```

Solution:

data	first	last	pidx	rand	proc
	0	9			sort
{4,2,3,1,0,5,6,8,7}	0	9	5	0	partition
	0	5			sort
{0,2,3,1,4,5,6,8,7}	0	5	4	0	partition
	0	4			sort
{0,2,3,1,4,5,6,8,7}	0	4	0	0	partition
	0	0			sort
	1	4			sort
{0,1,2,3,4,5,6,8,7}	1	4	2	0	partition
	1	2			sort
	3	4			sort
	5	5			sort
	6	9			sort
{0,1,2,3,4,5,6,8,7}	6	9	6	0	partition
	6	6			sort
	7	9			sort
{0,1,2,3,4,5,6,7,8}	7	9	8	0	partition
	7	8			sort
	9	9			sort