CS/Math 240: Supplemental Handout on Selection Sort

Roger Jenke

3/05/2011

1 Introduction to Selection Sort

Consider the following program specification:

Input: An integer $n \geq 0$ and an array $A[0..n-1]$ of integers.
Output: The array A sorted from smallest to largest.

We claim that the following implementation, known as selection sort, correctly matches the above specification.

```
SelectionSort(n, A)
(1)   i ← 0
(2)   while i < n
(3)       m ← i
(4)       j ← i + 1
(5)       while j < n
(6)           if A[j] < A[m] then m ← j
(7)           j ← j + 1
(8)       swap A[i] and A[m]
(9)       i ← i + 1
```

Running through selection sort on a sample array, say $[4, 3, 5, 2, 1]$, we can see that the algorithm is in fact correct on this specific input. In order to observe the function of each pointer i, j, and m, we will denote their position in the array with the corresponding subscript at each iteration. For example, at initialization $i \leftarrow 0$, so the array will be denoted as $[4_i, 3, 5, 2, 1]$, with i's pointer at $A[0]$. The algorithm proceeds as follows:

$i \leftarrow 0$

$[\ 4_{i,m}\ \ 3_j\ \ 5\ \ 2\ \ 1]$
$[\ 4_i\ \ 3_{m,j}\ \ 5\ \ 2\ \ 1]$ $\qquad j \leftarrow 1; A[1] < A[0] \Rightarrow m \leftarrow 1$
$[\ 4_i\ \ 3_m\ \ 5_j\ \ 2\ \ 1]$ $\qquad j \leftarrow 2; A[2] > A[1]$
$[\ 4_i\ \ 3\ \ 5\ \ 2_{m,j}\ \ 1]$ $\qquad j \leftarrow 3; A[3] < A[1] \Rightarrow m \leftarrow 3$

$[\ 4_i\ \ 3\ \ 5\ \ 2\ \ 1_{m,j}]$ $\quad\quad$ $j \leftarrow 4; A[4] < A[3] \Rightarrow m \leftarrow 4$
$[\ \mathbf{1}_i\ \ 3\ \ 5\ \ 2\ \ \mathbf{4}_{m,j}]$ $\quad\quad$ $SWAP(0,4)$

$i \leftarrow 1$

...
$[\ 1\ \ 3_i\ \ 5\ \ 2_m\ \ 4_j]$
$[\ 1\ \ \mathbf{2}_i\ \ 5\ \ \mathbf{3}_m\ \ 4_j]$ $\quad\quad$ $SWAP(1,3)$

$i \leftarrow 2$

...
$[\ 1\ \ 2\ \ 5_i\ \ 3_m\ \ 4_j]$
$[\ 1\ \ 2\ \ \mathbf{3}_i\ \ \mathbf{5}_m\ \ 4_j]$ $\quad\quad$ $SWAP(3,4)$

$i \leftarrow 3$

...
$[\ 1\ \ 2\ \ 3\ \ 5_i\ \ 4_{j,m}]$
$[\ 1\ \ 2\ \ 3\ \ \mathbf{4}_i\ \ \mathbf{5}_{j,m}]$ $\quad\quad$ $SWAP(4,5)$

....

$i \leftarrow 5$; **RETURN**

Of course, showing that the algorithm is correct on a single input is not nearly sufficient for proving general correctness. However, observing the behavior of SelectionSort on a typical input allows us to better understand the role of each line and variable of the algorithm. From our observations, we note that each iteration of the main loop consists of two parts. First, the nested loop executes in a manner that ends with j at the end of the array, and m at the index whose corresponding element is minimal in the array, starting from position i. Then, the elements in positions m and i are swapped.

Again, these observations are not sufficient for proving general correctness! In order to do so, it is necessary to formalize our reasoning; this ensures that our intution about SelectionSort is in fact correct for all possible inputs to the algorithm.

Our usual approach to proving program correctness when dealing with programs that loop is to introduce an invariant which will hold at each iteration, and then to use that invariant to show both partial correctness and termination. Examining the behavior of SelectionSort on our example array might even suggest such an invariant [hint: what can be said about the array elements A[0...i] after each iteration?]. However, the main issue with directly introducing an invariant on i for SelectionSort is that reasoning about this invariant will be complicated by the existence of a nested loop.

Let's say that we are able to determine an invariant for SelectionSort. This invariant will be expressible in the form, "$\forall k \geq 0$, after k iterations of the main loop, a certain property of A will

hold." Establishing a base case for this invariant may be simple, as it would be concerned only with A's intial conditions. However, our inductive step requires that, given k correct iterations of our main loop, we can argue the correctness of the (k + 1)th iteration. This iteration itself contains a nested while loop whose behavior can vary depending on the size and contents of A. In order to reason about an iteration of our main loop, then, it is necessary to understand the behavior of the nested loop within it.

Earlier, we described the function of the nested loop generally in terms of the variables j and m. In fact, we note that the important result of the loop is more specifically concerned with only m. We would like to able to state that whenever this loop executes on a given range of A, on termination m is set to the minimum value within the range. That is, we want the nested loop to meet the following specifications.

Input: An integer $n \geq 0$, an integer $0 \leq i < n$ and an array $A[0..n-1]$ of integers.
Output: The index of the smallest element of the array A within the range i,...n-1.

In order to prove that our nested loop does indeed match our desired specifications, it is necessary to prove the loop's correctness. To do this, we recast our loop as its own subalgorithm, which we will refer to as FindMin.

1.1 The FindMin Algorithm

We now formally introduce our 'new' algorithm, which we will call FindMin. FindMin proceeds as follows:

```
FindMin(i, n, A)
(1)   m ← i
(2)   j ← i + 1
(3)   while j < n
(4)       if A[j] < A[m] then m ← j
(5)       j ← j + 1
```

We have noted previously that FindMin will produce the index of the smallest element from the subarray $A[i...n-1]$. Examining the algorithm, we can see more specifically why this holds. During execution, the variable j iterates through the subarray. The variable m serves as a pointer to the minimal element encountered thus far; if an element $A[j]$ is found to be smaller than $A[m]$, the value of m is updated accordingly. In order to state this observiation formally, we will introduce the following invariant.

Invariant A. *$i < j \leq n$, $i \leq m < j$, and $A[m]$ is the smallest element from $A[i...j-1]$.*

We prove that this invariant holds after any number of iterations through induction. The base case for this invariant, P(0), holds trivially; we have $i < n$ from our program specification, and we intialize $j \leftarrow i+1$ and $m \leftarrow i$. These satisfy our first two conditions. The subarray $A[i....(i+1)-1]$ is simply $A[i]$, so $A[i]$ is the smallest element. This matches our assignment of m.

Assuming P(t), that the loop invariant holds after t iterations, we must show that P(t+1) holds as well. We start by noting that our invariant on the values of j and m will hold in all cases. The value of j always increases by one at the end of each iteration of the loop; further, the value of m only changes when it is set to j, which occurs before this increase. This establishes that the lower bounds of both variables is maintained from $P(t) \rightarrow P(t+1)$, and also that $m < j$ at each iteration. $j \leq n$ is established since the loop has entered the (t+1)th iteration; otherwise, the while condition would not have held, and the iteration would not have taken place.

We now have $i < j \leq n$, $i \leq m < j$. It remains to be shown that $A[m]$ is the smallest element from $A[i...j-1]$ at iteration (t+1). This is accomplished through a proof by cases.

Case One: $A[j] < A[m]$

> We denote the inital values of m and j during this iteration as m_t and j_t, from the inductive hypothesis, we know that $A[m_t]$ is the smallest element from $A[i....j_t - 1]$. If $A[j_t] < A[m_t]$, then it is is also smallest than any element within the previously stated range; so it is the smallest element from $A[i...j_t]$. Since $m \leftarrow j$ after the if statement is triggered, and we know that $j_{t+1} = j_t + 1$, we have $A[m]$ as the smallest element from $A[i...j_{t+1} - 1]$. So our invariant holds.

Case Two: $A[j] \geq A[m]$

> In this case, $A[m_t]$ is smaller than all other elements from $i....(j_t - 1)$, and is smaller than or equal to $A[j_t]$. In either case, $A[m_t]$ remains the smallest element from $A[i....j_t]$, with $j_t = j_{t+1} - 1$; m is unchanged in this iteration, so the invariant will hold.

Partial correctness for FindMin follows directly from our invariant. In order for our while loop to terminate, we must have $j \geq n$. But from our invariant, after any amount of iterations $j \leq n$. This gives the strictly equality $j = n$. Substituting n into our invariant, $A[m]$ must be the minimum element from $A[i....(n-1)]$. So if FindMin returns a value m, it will match our program specifications, proving correctness.

Termination for FindMin can be shown by observing that the counter j increases by 1 with each iteration, until a finite n is reached. Since we know that initially $j \leftarrow i$, and that when the while loop terminates, $j = n$, an exact number of iterations for FindMin can actually be given as $n - i$.

1.2 Correctness of Selection Sort

We can now rewrite our SelectionSort algorithm using FindMin as a subalgorithm. The new definition is as follows:

```
SelectionSort(n, A)
(1)   i ← 0
(2)   while i < n
(3)       m ←FINDMIN(i, n)
(4)       swap A[i] and A[m]
(5)       i ← i + 1
```

This condensed algorithm is much easier to reason about, and now we can introduce the invariant for the main loop that was alluded to in the introduction.

We again observe the behaviour of our remaining variables m and i during the example run of SelectionSort. Note that during the first iteration, the smallest value in the array is swapped to position 0; then, in the second iteration the smallest value in the remaining array is swapped into the next position (1), etc. This has the effect of sorting the array from left to right; more formally, this behavior can be captured by the following invariant.

Invariant B. $0 \leq i \leq n$, *Each element in* $A[0...i-1]$ *is less than or equal to every element of* $A[i...n-1]$, *and* $A[0...i-1]$ *is in sorted order.*

P(0) holds trivially; since $A[0...-1]$ is the empty array, it has no elements and the second half of the invariant holds. Initially $i \leftarrow 0$; since $n \geq 0$ from our specification, the first half of the invariant holds as well. To outline our inductive argument, we also establish P(1) [note that this step is not strictly necessary to prove correctness]. After a single iteration of the loop, FindMin(0, n) will set m equal to the smallest element in the full array; then, this element will be swapped to position 0. So the subarray $A[0....0]$, or just $A[0]$, will contain the 1 smallest element of A, which is trivially sorted among itself.

Assuming P(t), it is necessary to prove P(t + 1). At the start of the (t+1)th iteration $i = i_t$, so FindMin(i_t, n) will set m to the smallest element in the subarray $A[i_t...n-1]$. From the inductive hypothesis, all of the elements $A[0...(i_t-1)]$ are smaller than or equal to any element in $A[i_t+1...(n-1)]$. After the swap, $A[i_t]$ will also satisfy this condition; so when $i_{t+1} = i_t + 1$ at the end of the loop, each element of $A[0....i_{t+1}-1]$ will be less than or equal to the elements of $A[i_{t+1}....n-1]$.

From our inductive hypothesis, A[m] cannot be smaller than any of the elements from $A[0...i_t]$. So when $A[m]$ is swapped with $A[i_t]$, the elements $A[0...i_t]$ will be properly sorted among themselves ($A[0....i_t-1]$ is sorted from the IH, and a larger element is placed at its end). Thus $A[0....i_{t+1}-1]$ is sorted at the end of the iteration, both conditions of our invariant are satisfied, and the induction step is complete.

Invariant B suffices to show that the array A will be sorted if SelectionSort terminates; however, this is not enough to prove the correctness of the algorithm. For example, imagine an algorithm that replaces all elements of A with the integer 1. This array will be sorted on termination; however, it will not return the desired result (which is a sorted version of the original array). Specifically for SelectionSort, we can see that if the variables i and m ever leave the bounds of the original array, it is possible to change the overall contents of A during a swap, so that the program returns an incorrect array. To ensure that SelectionSort maintains the output specifications given in the introduction, then, it is useful to add the following to our invariant:

Invariant B_2. *A is a permutation of the original input array; that is, there are no elements of A that were not in the original array (and vice versa)*

P(0) holds for this invariant, as no swap operations have occured. Assuming P(t) holds, we must show that P(t+1) holds as well. From Invariant A (from the FindMin algorithm), we know that $i \leq m < n$; and Invariant B states that $0 \leq i \leq n$ at the end of iteration (t+1). Since the swap operation occurs before i is incremented, we then have $0 \leq i < n$ at the swap. So the elements that are swapped still both exist within the range of $A[0...n-1]$; thus our permutation condition holds for P(t+1).

Partial correctness for SelectionSort now directly follows from our two invariants; similarly to our FindMin argument, when SelectionSort terminates, $i \geq n$, and from our invariant $i \leq n$, so $i = n$. Thus the elements $A[0...n-1]$ (in other words, the full array) are sorted. This is the full array A; and from invariant B_2, A contains the same elements as our original input array. So our output matches the original specifications, and SelectionSort is partially correct.

Termination can again be shown by observing that the counter i increases by 1 with each iteration. Note that an exact number of iterations can be given for SelectionSort's main loop as well; combined with an analysis of the time spent on each FindMin call (which varies with i), an exact value for the total number of loop iterations in SelectionSort can be derived. This is left as an exercise for the curious reader.