CS1112-202/206/207–Spring 2010

Section 13 Solution

Friday, April 30

1 Insertion Sort

First, we recall the method of insertion sort. The main idea of insertion sort is that, throughout its the execution, the algorithm maintains the original vector in two parts: sorted and unsorted. The algorithm starts with the sorted part empty and unsorted part containing the original vector. The goal is to remove one element from the unsorted part at a time and add it to the sorted part so as to keep the sorted part sorted. The algorithm is done when the unsorted part is empty.

Refer to the functions at the bottom of the first page of the exercise sheet. `InsertionSort` is the algorithm mentioned above, and `Insert` is the procedure of "adding an element to the sorted part to keep the sorted part sorted." In each iteration of `InsertionSort`, the sorted part is `x(1:k-1)` and the unsorted part is `x(k:n)`. (Note that k starts from 2 because any length-1 vector is always sorted.) The algorithm takes the next element in the unsorted part (in this case, it is always `x(k)`) and put it in the sorted part, using `Insert` to maintain the sortedness of the sorted part. Once this is done, `x(1:k)` becomes the new sorted part, and we can move on to the next k.

Understanding the idea, we see that there is no problem not using a subfunction—`Insert`—so that we can avoid creating a new vector. Furthermore, `InsertionSortInplace` does not require us to keep track of the numbers of comparisons and swaps. Therefore, we can rewrite `InsertionSort` to obtain `InsertionSortInplace` as follows:

```
function x = InsertionSortInplace(x)
% Sort x in ascending order using the insertion sort algorithm.
% Sort in-place, i.e., without creating another vector.
% Perform the insert process in-line, i.e., no subfunction.
% x is a column vector.

n=length(x);
for k=2:n
    i=k-1;  % i is k in Insert
    while i>=1 && x(i)>x(i+1)
        tmp=x(i+1);     % tmp is t in Insert
        x(i+1)=x(i);
        x(i)=tmp;
        i=i-1;
    end
end
```

The following is not required to understand the solution, but if you are interested, read on.

Observe that in the solution above, we have to swap the value we want to insert with the values greater than it until the value being inserted gets into its correct place. This takes three assignments

per swap. This can be improved as follows. First, note that after we are done swapping for each element *e* being inserted, we effectively "slide" all the elements whose values are greater than *e* one position to the right, and then we simply put *e* in the now-vacant position. To reflect this idea, the body of the `for` loop can be rewritten as

```
e=x(k); % e is the element from the unsorted part we want to insert
i=k-1;  % i+1 is always the vacant slot (which is k originally)
        % in this case we are making slot k vacant
while i>=1 && x(i)>e
    x(i+1)=x(i);  % move x(i) to the right
                  % so the vacant slot moves left
    i=i-1;
end
x(i+1)=e;   % i+1 is now the final vacant slot; put e here
```

2 Merge Sort

There are four lines that can produce output:

- `n=length(x)`
- `yL=mergeSort(x(1:m))`
- `yR=mergeSort(x(m+1:n))`
- `y=merge(yL,yR)`

because there is no semicolon at the end of each of these lines.

If $n = 1$, then only the first line above produces one line of output; otherwise, each of the four lines produces one line of output. First, let us speculate what the body of `mergeSort` should display when it is called with `a`. Since the length of *a* is 10, we expect that it will print out four lines. The value of a line is printed *after* all the execution on that line is done. That is,

- 10—the value of `n`—will be printed when `length(x)` (which gives 10) is done executing,
- The value of `yL` will be printed when `mergeSort(x(1:m))` is done executing,

and so on. However, when the recursive call to `mergeSort(x(1:m))` is executing, it also prints to the screen. Therefore, before `mergeSort(a)` outputs its second line, there will be output from `mergeSort(x(1:m))` displayed on the screen.

We need to repeat this argument to each recursive call of `mergeSort`*!* Figure 1 shows the (decorated) output of executing the two statements on the exercise sheet. `SORT` refers to a call to `mergeSort`, but the argument to `SORT` refers to the *actual* value being passed in that call, totally for your reference. The indentations indicate which line comes from which call. The actual output can be obtained from removing all the decorations (lines, codes, and indentations) from the figure.

Note that the figure leaves out the variable names (which should also be displayed along with their values), but you can figure this out by matching the name of the variables with their corresponding recursive call and the order in which they are printed in that particular call.

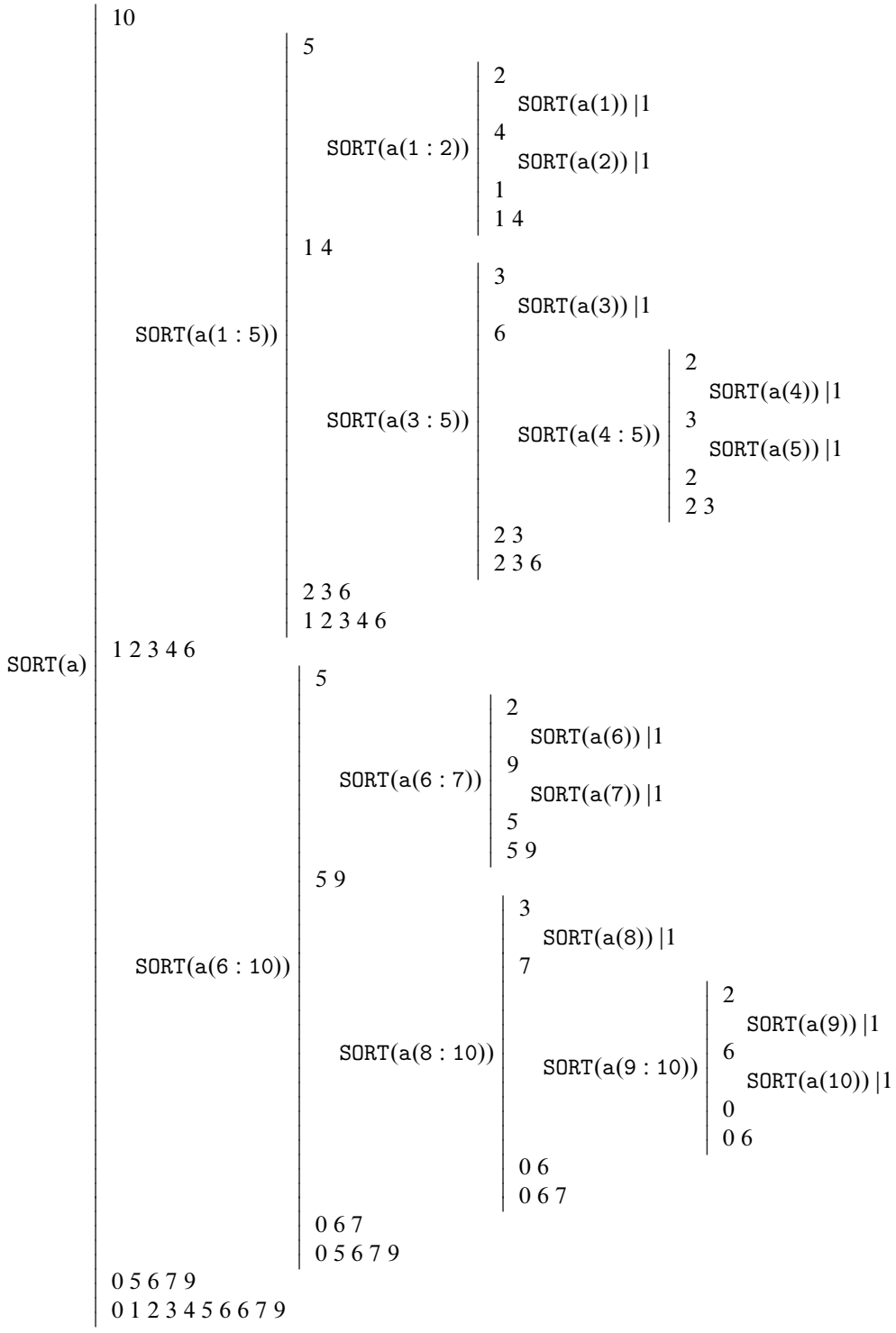

Figure 1: The output when executing `mergeSort(a)`, where `a` is as defined on the exercise sheet.